16-833: Robot Localization and Mapping, Fall 2024

Homework 1
Robot Localization using Particle Filters

Jinjiang You (jinjiany), Yixin Fei (yixinf).

1 Approach and Implementation

In this homework, we implement Monte Carlo Localization (MCL), an application of particle filtering for robot localization. The localization using particle filters gets rid of the parametric distribution and turns it into a sample-based representation. The key idea of the particle filter is to represent the posterior $bel(x_t)$ by a set of random state samples drawn from this posterior. First, we use the motion model as the prediction step to estimate where the robot is now by using the odometry command and the previous state. Then, we use the sensor model as the correction step to compute the importance weights for each particle. Finally, we resample particles with probabilities proportional to their importance weights.

1.1 Motion Model

1.1.1 Description

Particle filter represents the posterior $bel(x_t)$ by a set of random state samples drawn from this posterior. What motion model does is predict, or sample the particle states using the odometry command and the previous state.

1.1.2 Implementation

Following [1], we choose to implement the Odometry Motion Model instead of the Velocity Motion model because we have the odometry measurements. The algorithm is described in Algorithm 1.

Given the odometry readings of two consecutive time $(\overline{x}, \overline{y}, \overline{\theta})$ and $(\overline{x}', \overline{y}', \overline{\theta}')$, the relative difference between them can be illustrated by a combination of the first rotation δ_{rot1}, the translation δ_{trans}, and the second rotation δ_{rot2}. The measured rotation and translation are computed through Line 1 - Line 3. There is uncertainty in the measurements, so we need to model how the noise affects these parameters. The true values of the rotation and translation are obtained from the measured ones by subtracting the noise (Line 4 - Line 6). The noise is implemented by random sampling from a normal (Gaussian) distribution with zero mean and variance b^2. Finally, the true state (x', y', θ') is obtained from the previous state (x, y, θ) by adding the true values of the rotation and translation (Line 7 - Line 9).

The error parameters α_1, α_2, α_3, and α_4 are robot-specific parameters that control the noise in robot motion. Usually, the larger the motion we ask the robot to

move, the higher the error is. α_1 measures how the rotation affects the rotation. α_2 measures how the translation affects the rotation. Similarly, α_3 is the translation error on translation, and α_4 is the rotation error on translation. After adjusting the sensor model to converge to the correct region of the map, we increase the value of α a bit to introduce appropriate noise and obtain a banana-shaped distribution.

Algorithm 1 Motion Model

Input: $u_{t-1} = (\overline{x}, \overline{y}, \overline{\theta}), u_t = (\overline{x}^{'}, \overline{y}^{'}, \overline{\theta}^{'}), x_{t-1} = (x, y, \theta)$
Output: $x_t = (x^{'}, y^{'}, \theta^{'})$

$\delta_{rot1} = atan2(\overline{y}^{'} - \overline{y}, \overline{x}^{'} - \overline{x}) - \overline{\theta}$
$\delta_{trans} = \sqrt{(\overline{x} - \overline{x}^{'})^2 + (\overline{y} - \overline{y}^{'})^2}$
$\delta_{rot2} = \overline{\theta}^{'} - \overline{\theta} - \delta_{rot1}$
$\hat{\delta}_{rot1} = \delta_{rot1} - \textbf{sample}(\alpha_1\delta_{rot1}^2 + \alpha_2\delta_{trans}^2)$
$\hat{\delta}_{trans} = \delta_{trans} - \textbf{sample}(\alpha_3\delta_{trans}^2 + \alpha_4\delta_{rot1}^2 + \alpha_4\delta_{rot2}^2)$
$\hat{\delta}_{rot2} = \delta_{rot2} - \textbf{sample}(\alpha_1\delta_{rot2}^2 + \alpha_2\delta_{trans}^2)$
$x^{'} = x + \hat{\delta}_{trans}\cos(\theta + \hat{\delta}_{rot1})$
$y^{'} = y + \hat{\delta}_{trans}\sin(\theta + \hat{\delta}_{rot1})$
$\theta^{'} = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$
return $x_t = (x^{'}, y^{'}, \theta^{'})^T$

1.2 Sensor Model

1.2.1 Description

Following the motion model, the sensor model computes the importance weights for all particle. It takes the laser measurements, compares them with the ground truth values (which are computed by ray casting algorithm), and return a probability for each particle.

1.2.2 Ray Casting Implementation

The sensor model needs the ground truth depth measurements to compute the probability. Therefore, we implemented the Ray Casting algorithm, which takes the ground truth map, the position of the robot, and the orientation of the robot as inputs, and computes the depth measurements for the robot at that pose.
The ray casting algorithm for a single ray is quite simple. Given a ray origin $\mathbf{p}_0 = (x_0, y_0)$, ray direction $\mathbf{d} = (\cos\theta, \sin\theta)$, and an initial marching length $l = 0$. We repeatedly add a constant s (we call it marching step. Usually we set it to be the map resolution) to l, and compute the ray target position $\mathbf{p} = \mathbf{p}_0 + l\mathbf{d}$. Then we check whether the position of $\mathbf{p}$ is occupied or not, using the occupancy map. If it is occupied, the current value of l is the ray casting result. Otherwise, we continue this process, until some terminating condition is met (e.g. we can set a maximum value for the marching length).

Improvement 1: This algorithm is slow because we need to do it for every ray. Instead, **we implemented a vectorized algorithm using Python**. Theoretically, it can perform ray casting for any number of rays. In practice, we fix the ray origin and perform the ray casting process for 360 rays each time.

Algorithm 2 Ray Casting for a single ray

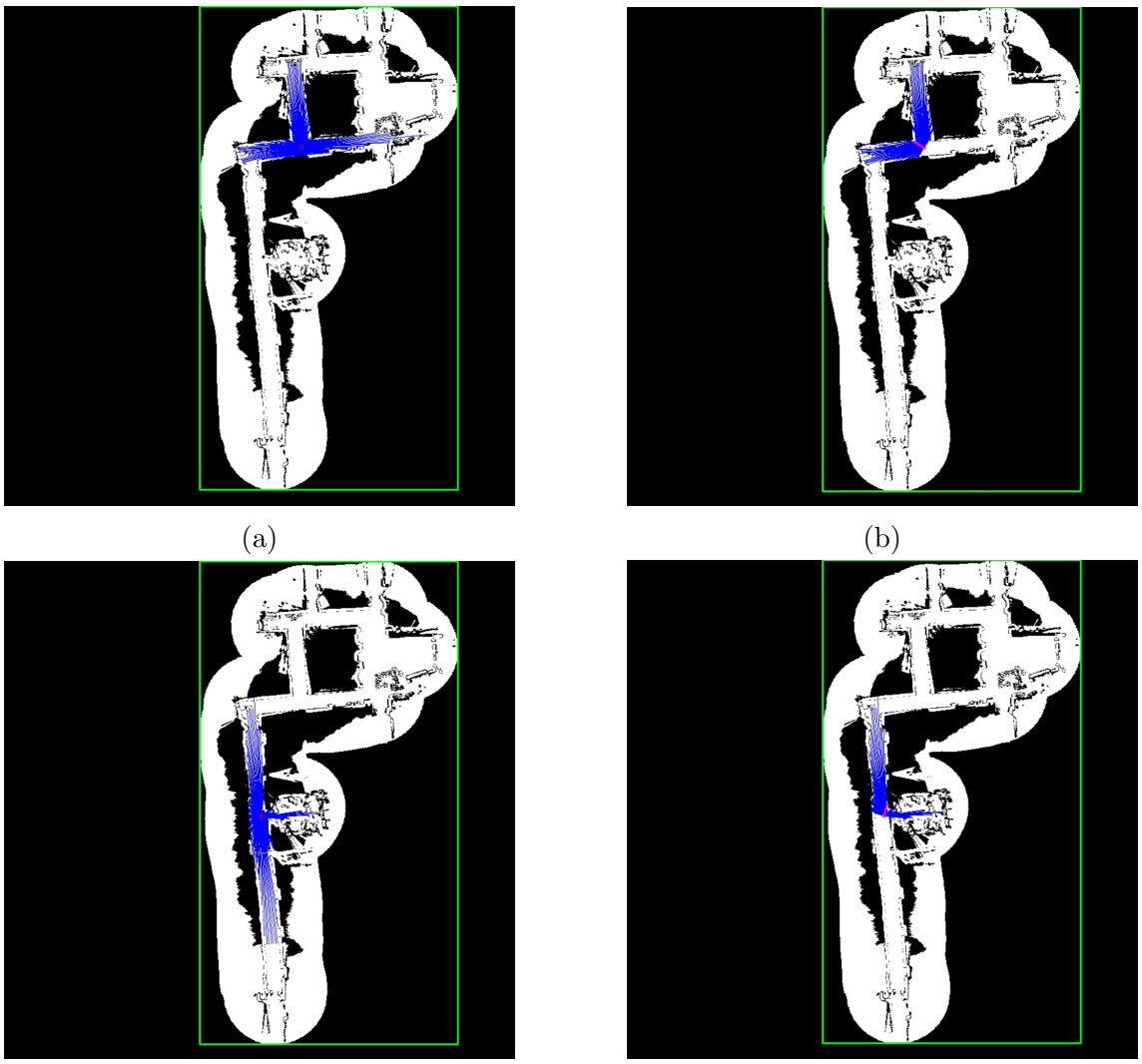
Input: ray origin $\mathbf{p}_0 = (x_0, y_0)$, ray direction $\mathbf{d} = (\cos\theta, \sin\theta)$, map M, marching step s, terminating length t
Output: ray length L

$l = 0$
while $l \leq t$ **do**
    $\mathbf{p} = \mathbf{p}_0 + l\mathbf{d}$
    if $M(\mathbf{p})$ is occupied **then**
        return l
    end if
    $l = l + s$
end while

Algorithm 3 Ray Casting for multiple rays

Input: ray origin ray_origin $= (x_0, y_0)$, map M, marching step s, terminating length t
Output: ray lengths L $= [L_0, L_1, \ldots, L_{359}]$

$l = 0$
L $= [0, 0, \ldots, 0]$
indices $= [0, 1 \ldots, 359]$
ray_dir $= [[\cos i\pi/180, \sin i\pi/180]$ **for** i **in range**$(360)]$
while indices.shape[0] > 0 and $l \leq t$ **do**
    p $=$ ray_origin $+ l \cdot$ ray_dir [indices]
    hit $= (M$ [p[1]] , [p[0]] == occupied)
    L [indices [hit]] $= l$
    indices $=$ indices [**not** hit]
    $l = l + s$
end while
L [indices] $= t$
return L



(a) (b) (c) (d)

Figure 1: Ray casting results visualization. Left: Ray casting 360 degrees. Right: Ray casting 180 degrees with specified orientations.

Improvement 2: In fact, the ray casting result can be pre-computed and stored in a file. Once we need it, we can load it from the file and directly read data from it. This is very useful because it makes our debugging faster.

We implemented a class called *RayCastingResult* in the file *ray_casting_result.py*. It can compute ray casting results for the whole map, storing them into a file, or loading from a file.

Improvement 3: Actually, we do not need to perform ray casting at all locations on the map. Only a small part of the map is valid. This fact can help us reduce computation time and file size.

Finally, we implemented a visualizer in *ray_casting_result.py* to visualize the ray casting result, shown in figure 1. The ray casting result file is available on google drive. Download it to your machine, and set the *ray_casting_result_path* argument to the path to it, and run *ray_casting_result.py*. You can click the left mouse button to set the robot's location, and click the right mouse button to set robot's orientation.

1.2.3 Sensor Model Implementation

The aim of the sensor model is to compute the probability $p(z_t|x_t, m)$. Because we collect the data from the laser range which measures the range to nearby objects, we choose the beam model. After calculating the true range of the object using the ray casting, we implement the density with the combination of four different models following Chapter 6.3 of [1].

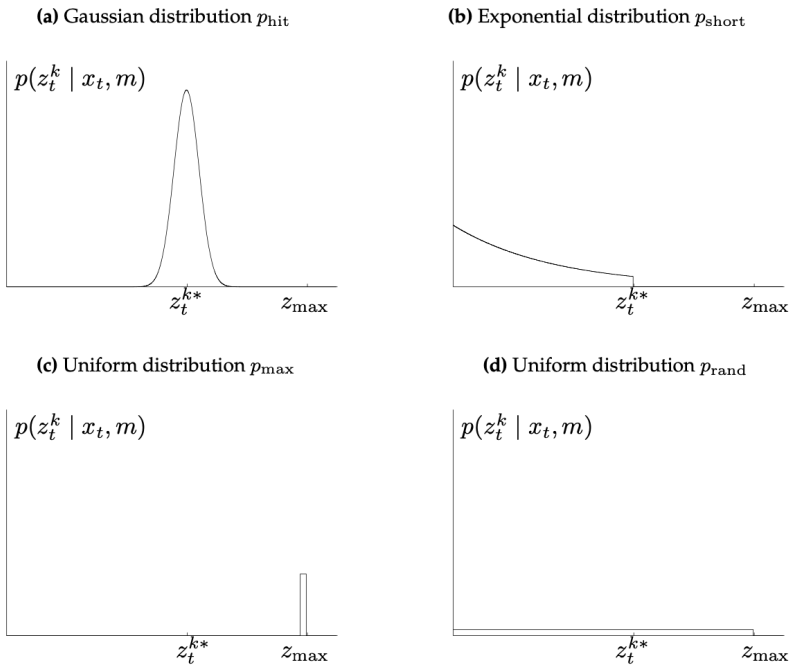


Figure 2: Four different probability models.

1. **Measurement Noise.** Even though the obstacles are expected, the measured values are not ideally accurate all the time. Sometimes the sensor becomes incorrect due to the noise. The probability of getting a certain measurement can be described by the Gaussian distribution. The mean of the Gaussian distribution is the expected measurement z_t^{k*} and the standard deviation is σ_{hit}, as shown in (a) of 2. We implement the measurement noise probability by:

$$p_{hit}(z_t^k|x_t, m) = \begin{cases} \eta\mathcal{N}(z_t^k; z_t^{k*}, \sigma_{hit}^2) & if\ 0 \le z_t^k \le z_{max} \\ 0 & otherwise \end{cases}$$

$$\mathcal{N}(z_t^k; z_t^{k*}, \sigma_{hit}^2) = \frac{1}{\sqrt{2\pi\sigma_{hit}^2}} e^{-\frac{1}{2}\frac{(z_t^k - z_t^{k*})^2}{\sigma_{hit}^2}}$$

2. **Unexpected Objects.** In the real scene, unexpected obstacles may pop up and obstruct the field of view. This can be described by an exponentially decaying function in (b) of 2, which starts high, then decays, and stops at the expected measurement z_t^{k*}. We only focus on the range between 0 and z_t^{k*} because we only care about the dynamic obstacles between us and the obstacles we are expected to see. The reason that the probability is exponentially decaying is that only the closest obstacle to the scanner should be considered. When the first closest obstacle appears, the objects behind it do not matter

anymore. We implement the unexpected objects by:

$$p_{short}(z_t^k|x_t,m) = \begin{cases} \eta\lambda_{short}e^{-\lambda_{short}z_t^k} & if\ 0 \le z_t^k \le z_t^{k*} \\ 0 & otherwise \end{cases}$$

3. **Failures.** Failures may happen and the sensor does not return anything when there is a total reflection or bright sunlight. The range of the laser is also limited. When there is a large open wall, the sensor fails to detect it and obstacles are missed. This case is modeled as (c) in 2.

$$p_{max}(z_t^k|x_t,m) = \begin{cases} 1 & if\ z = z_{max} \\ 0 & otherwise \end{cases}$$

4. **Random Measurements.** Something we haven't modeled before but randomly happens, so we use a mall uniform probability to illustrate it, as shown in (d) of 2.

$$p_{rand}(z_t^k|x_t,m) = \begin{cases} \frac{1}{z_{max}} & if\ 0 \le z_t^k \le z_{max} \\ 0 & otherwise \end{cases}$$

Finally, the mixture density can be obtained by adding the four different distributions together.

$$p(z_t^k|x_t,m) = \begin{pmatrix} z_{hit} \\ z_{short} \\ z_{max} \\ z_{rand} \end{pmatrix} \cdot \begin{pmatrix} p_{hit}(z_t^k|x_t,m) \\ p_{short}(z_t^k|x_t,m) \\ p_{max}(z_t^k|x_t,m) \\ p_{rand}(z_t^k|x_t,m) \end{pmatrix}$$

We find that using a summation of log likelihoods to replace the multiplication of probabilities can reduce the numerical error. z_{hit}, z_{short}, z_{max}, and z_{rand} are four parameters that control the weight of each distribution. We tune these parameters to make the model converge to the correct initial place. We increase the z_{rand} so that the model is more likely to converge when there is much noise in the real world. For more information, see 3.

1.3 Resampling

1.3.1 Description

The resampling is a procedure that replaces unlikely samples with more likely ones. It makes sure that the particles after resampling process follow the probability distribution computed by the sensor model.

1.3.2 Implementation

We implement two sampling strategies. First, we sample from a multinomial distribution using importance weights of all particles. However, this strategy may introduce a high variance. Therefore, we utilize the low variance sampling illustrated by [1]. It is also called stochastic universal resampling. Instead of choosing M random numbers and selecting those particles that correspond to these random numbers, this strategy computes a single random number and selects samples according to this number but still with a probability proportional to the sample weight. The length of each bucket is different, representing the weight of particles. We need only randomly choose one single number representing the first array and all the other arrays have equal space.

Algorithm 4 Low Variance Resampling

Input: $u_{t-1} = (\overline{x}, \overline{y}, \overline{\theta}), u_t = (\overline{x}', \overline{y}', \overline{\theta}'), x_{t-1} = (x, y, \theta)$
Output: $x_t = (x', y', \theta')$

$\overline{\chi}_t = \emptyset$
$r = rand(0; M^{-1})$
$c = w_t^{[1]}$
$i = 1$
$m = 1$
while $m \leq M$ **do**
    $U = r + (m-1) \cdot M^{-1}$
    while $U > c$ **do**
        $i = i + 1$
        $c = c + w_t^{[i]}$
    end while
    add $x_t^{[i]}$ to $\overline{\chi}_t$
    $m = m + 1$
end while
return $\overline{\chi}_t$

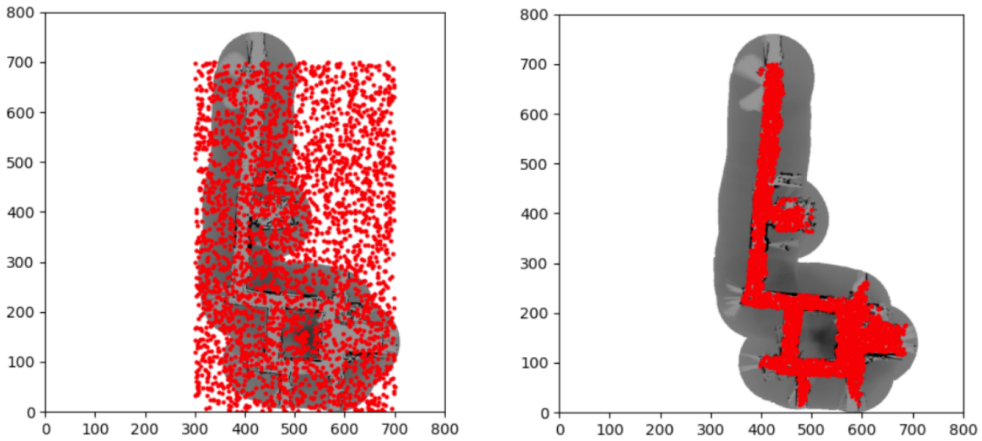
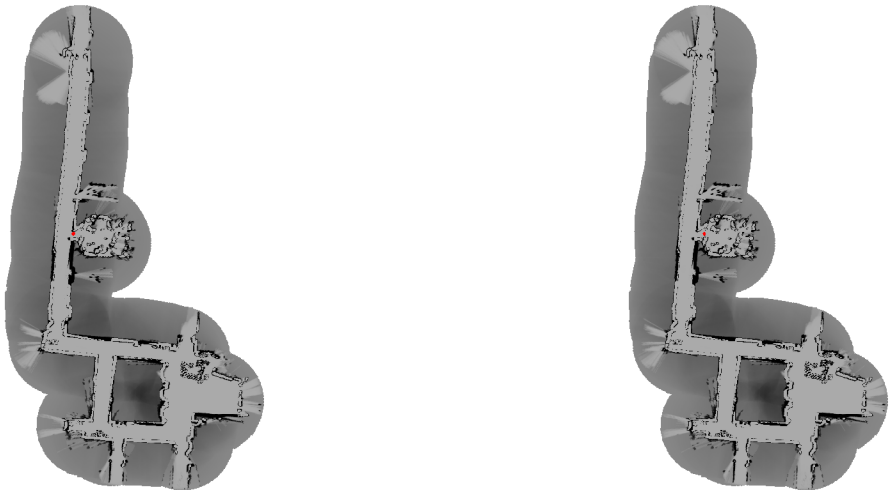


Figure 3: Initializing particles with two different strategies.

1.4 Initialization

Instead of simply initializing the particles using a regular rectangle area, we implement the *init_particels_freespace* function to initialize the particles in the hallway of the map, the valid space for robots. The original initialization includes the walls, obstacles, and places outside the map, which are inaccessible to the robot. Initializing particles only in unoccupied areas makes the convergence faster and reduces the computation. The left image shows that the initialized particles are everywhere on the map, and the right image shows only initializing the hallway of the map.



(a) (b)

Figure 4: Left: The last frame of log1. Right: The last frame of log2.

2 Experiments

2.1 Discussion of Performance

We precomputed the ray casting results and stored them on disk. We pick one ray origin and 360 ray directions for each map grid and only consider those "valid" map grids similar to 1.4. It takes about 20 minutes to compute the results and the file size is about 800MB.
Once the ray casting result is computed, the particle filter process is fast. It takes about 2 seconds to process a frame (for "L" laser data entry) when the particle number is 5000. For data *robotdata1.log*, it takes about 30 minutes to finish the whole algorithm pipeline. For data *robotdata2.log*, it takes about 90 minutes to finish the whole algorithm pipeline.

2.2 Robustness

Our algorithm may fail if the particles are not converged to the correct location in the first few frames. However, as long as they converge to the correct location, our algorithm will compute the correct trajectory of the robot.

2.3 Results

We run our algorithm on log 1 and log 2. The results are videos available on google drive (log 1, log 2). The video is generated by OpenCV's *VideoWriter*, with FPS equal to 60.

3 Repeatability

To reproduce the algorithm, you need to use the following parameters. The number of particles is 5000. The values of parameters we tune for the motion model and the sensor model are listed below.
We use relatively small values for these 4 parameters. Although the odometry measurements are not accurate, the algorithm relies on them to know when the

parameter	α_1	α_2	α_3	α_4
value	0.00005	0.00005	0.0075	0.0075

Table 1: Parameters for Motion Model.

robot turns in different directions, which means it needs to sample more particles that have these directions.

parameter	z_{hit}	z_{short}	z_{max}	z_{rand}	σ_{hit}	λ_{short}
value	10	0.09	0.05	2000	100	0.1

Table 2: Parameters for Sensor Model.

We use a small value for z_{hit} and a large value for z_{rand} because we find that if we don't, the particles will quickly converge to the wrong positions in the first few frames. The reason is the occupancy map is not perfect and the laser measurements also contain much noise. We want our algorithm to converge slower, using the measurements of the first several frames to determine the right particle states. That's why we give z_{rand} a high value. Also, we give σ_{hit} a high value to tolerate more noise in the laser measurements. Furthermore, z_{max} is the smallest because we don't find some "max range" value in the robot log file, which means the laser may not follow the assumption of the sensor model.

4 Future Work

The motion model and sensor model can be implemented in C++, using CUDA. Since particle updates in these two stages are independent of each other, we can utilize the GPU to parallelize the two tasks. The ray casting algorithm can be implemented in C++ using CUDA too.

The parameters may not need to be fixed throughout the algorithm. We can adopt adaptive parameters. For example, in the first one hundred frames, the algorithm should have a large value for z_{rand} to avoid quickly converging to the wrong particle states. Once it converges to the correct particle states, it can reduce z_{rand} and increase z_{hit}.

The visualizer of the whole algorithm is still not perfect because all particles are drawn in red circles, making it unable to figure out how many particles there are in a cluster. We can use different colors to represent different particle densities to make the visualization more clear.

Furthermore, we can implement an automatic parameter tuner to find the best parameter for our algorithm. We can use an adaptive number of particles to help the model converge faster.

References

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.